Automated Agent Decomposition for Classical Planning

Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling

Automated Agent Decomposition for Classical Planning

Matthew Crosby
School of Informatics
University of Edinburgh
Edinburgh EH8 9AB, UK
m.crosby@ed.ac.uk

Michael Rovatsos
School of Informatics
University of Edinburgh
Edinburgh EH8 9AB, UK
mrovatso@inf.ed.ac.uk

Ronald P. A. Petrick
School of Informatics
University of Edinburgh
Edinburgh EH8 9AB, UK
rpetrick@inf.ed.ac.uk

Abstract

Many real-world planning domains, including those used in common benchmark problems, are based on multiagent scenarios. It has long been recognised that breaking down such problems into sub-problems for individual agents may help reduce overall planning complexity. This kind of approach is especially effective in domains where interaction between agents is limited.

In this paper we present a fully centralised, offline, sequential, total-order planning algorithm for solving classical planning problems based on this idea. This algorithm consists of an automated decomposition process and a heuristic search method designed specifically for decomposed domains. The decomposition method is part of a preprocessing step and can be used to determine the "multiagent nature" of a planning problem prior to actual plan search. The heuristic search strategy is shown to effectively exploit any decompositions that are found and performs significantly better than current approaches on loosely coupled domains.

1 Introduction

Recent work has shown how the multiagent structure inherent to certain planning domains can be exploited to improve planning times (Nissim, Brafman, and Domshlak 2010; Nissim, Apsel, and Brafman 2012). These approaches divide planning domains into sub-problems such that each agent can use only a subset of the available actions. This process creates a new problem, one of co-ordination among agents, which usually dominates search time. Therefore, the multiagent planning literature tends to focus on loosely coupled domains with minimal interaction between agents.

Most multiagent planning approaches require agents to be specified in advance. A human expert must work out the decomposition and also has to be familiar with a particular multiagent extension (beyond, e.g., standard PDDL). This raises the following question: Can we create an automated process to find multiagent decompositions of classical planning problems? In this paper we address this question, along with its natural follow-up: Can we exploit the structure afforded by such decompositions to develop faster planning algorithms?

In answer to the first question we present a novel, fully automated process for calculating agent decompositions of STRIPS-style planning problems. The decomposition algorithm utilises the multi-valued planning task (MPT) representation and causal graph generated by the Fast Downward planning system (Helmert 2006). By analysing the causal graph, it finds sets of variables with limited interactions between them, that form the basis of a decomposition. This analysis is performed as part of a static preprocessing phase and does not require state-space or plan-space search. As not all domains have a useful decomposition, the algorithm will sometimes report that no decomposition can be found and a single-agent search method can then be applied.

As we are mainly interested in fast, scalable plan computation, we focus on benchmark domains from the International Planning Competition (IPC). Here, we find domains which have very obvious multiagent decompositions (e.g. Satellites, Rovers) with little or no interaction between the agents. At the other end of the spectrum are domains for which no sensible decomposition exists (e.g. Sokoban[1], Visitall). However, there is also a middle ground where a decomposition can be found, but interaction between the subparts remains (e.g. Elevators, Airport). These constitute the most interesting cases for our method.

In answer to the second question, we propose a heuristic search algorithm which is based on the well-known "no delete lists" relaxation. This algorithm uses a novel approach to reduce the complexity of agent interactions. We present the results of this algorithm on the IPC domains and compare it to performant state-of-the-art planners, showing that it performs significantly better on a broad range of domains.

The contribution of this paper is twofold: Firstly, we present a fully automated decomposition algorithm that can be used to break down planning domains into subproblems with limited interaction and show that around a third of all IPC domains have such a decomposition. Secondly, we introduce a planning algorithm that exploits agent decompositions, and show that it greatly improves planning times.

The remainder of the paper is structured as follows: The next section discusses the related literature and Section 3 in-

[1] It is possible to modify Sokoban domains to be multiagent by adding additional 'player' objects. However, these are not present in the standard IPC problem set.

troduces an example problem that we will refer to throughout the paper. Section 4 introduces the required background from the literature and Section 5 formalises our framework. Sections 6 and 7 introduce algorithms for decomposition and search, respectively. Together, these form our planner (ADP). We present an empirical evaluation of our work in Section 8 and conclude with Section 9.

2 Related Work

The literature contains many different approaches related to multiagent planning. These include plan coordination (Cox and Durfee 2005) and plan merging/revision (Tonino et al. 2002) that assume post-planning coordination among individual agents, and also partial-order and/or concurrent planning (Yang, Nau, and Hendler 1992; Boutilier and Brafman 2001) and interleaved planning and execution (Brenner and Nebel 2009). In contrast, we follow the tradition of classical planning, i.e. we only consider centralised, offline, sequential, total-order planning in STRIPS-style planning domains.

The idea of using decompositions dates back to Lansky's early work (Lansky 1991), which proposed decomposing a domain into several potentially overlapping local search spaces (regions), and applying a hierarchical constraint satisfaction technique to combine them. However, this algorithm requires the user to provide explicit meta-knowledge about these regions in advance.

The idea of exploiting loose coupling among agents was further developed and refined by Brafman and Domshlak (2008). They investigate how to deal with "coordination points" in multiagent planning problems based on the distinction between public and private fluents. We build on this distinction in our work. Nissim et al's (2010) distributed multiagent planning algorithm also exploits loose coupling, solving a distributed CSP for those parts of the global plan where individual agents' contributions overlap.

While these approaches provide ways of analysing a domain in terms of how closely agents are coupled, this analysis can only be performed if the specification of the problems involves a set of agents from the outset. Our approach lifts this assumption by creating agents on demand.

In a similar vein, Nissim et al (Nissim, Apsel, and Brafman 2012) recently presented a similar automated method for partitioning planning domains. In their work, domains are decomposed based on a symmetry score which essentially maximises the number of internal actions (actions that do not affect other agents) each agent can perform. This is a rough approximation of how well decomposed a domain is based on the number of actions that can be pruned during search. They leave more accurate decomposition finding as an open challenge - one that we hope we contribute to.

In contrast to our heuristic approach they apply an optimal planning algorithm to the decomposed domains. Their algorithm has similarities with our search method that give insights into the fundamental properties of multiagent planning that allow for efficient search. Specifically, we share their idea to focus on single subgoals at a time and only look at subsequent actions that achieve that subgoal.

There are many other multiagent plan search algorithms in the literature that relate to the one we present: Ephrati and Rosenschein (1993) propose a mechanism where each agent votes for or against the next joint action in a multiagent plan based on whether local constraints are satisfied by a proposed state transition. Brafman et al (2009) apply constraint satisfaction techniques on so-called "agent interaction graphs" that specify which preconditions are needed from other agents for every action an agent intends to perform. Dimopoulos et al (2012) propose a multiagent extension of SATPLAN which models assistive actions through "external" preconditions that agents assume to be satisfied by other agents when needed. Our approach differs from these as it primarily aims at finding a valid plan fast rather than optimising against other criteria (e.g. game-theoretic stability, social welfare maximisation).

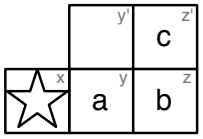

Figure 1: An example problem. The larger lower case letters represent robots that need to report to the starred square. The smaller lower case letters label the locations.

3 Example

As an example, consider a simple grid-world domain in which robots must report to a particular location without bumping into each other. A particular instance of this domain is shown in Figure 1. There are three robots (`a, b, c`) which must all report to the starred location. Robots can `move` to any orthogonal adjacent *empty* grid square and `report` at the intended destination. For example, if robot `a` ia at location `x` in can perform the report action to achieve the goal `report_a`.

This domain has a very obvious agent decomposition. If a human was asked to solve the problem, they would probably consider each robot's possible plans separately, and work out how to coordinate them. It is natural to attempt to find a plan for a to get to the goal, then move a out of the way. The next step is to find a plan for b to get to the goal and move it out of the way. Finally, a plan is found for c to reach the goal. Standard single-agent heuristics such as the no delete-lists heuristic will consider moving any robot closer to the goal an equally good action.

Our approach attempts to follow the human reasoning process as closely as possible. Firstly, our automated decomposition algorithm correctly separates the robots and leaves the grid-world itself as part of the public environment. Secondly, our planning algorithm proceeds to solve each individual robot's problem separately. This mimics the human approach outlined above including adding subgoals to move agents out of the way when necessary.

4 Background

The input for our planning algorithm is the *Multi-valued Planning Task (MPT)* (Edelkamp and Helmert 1999) representation calculated by the Fast Downward planner (FD) (Helmert 2006). This means we can solve problems written in propositional PDDL2.2, i.e. STRIPS domains with

arbitrary propositional formulae in preconditions and goal conditions, and conditional as well as universally quantified axioms and effects. For ease of presentation we assume a reduced form of MPTs without axioms and with conditional effects compiled away.

Definition 1 (Multi-valued planning tasks (MPTs)) *A (reduced)* multi-valued planning task (MPT) *is a 4-tuple* $\Pi = \langle V, I, G, A \rangle$ *where:*

- *V is a finite set of state variables v, each with an associated finite domain D_v,*
- *I is a full variable assignment over V called the initial state,*
- *G is a partial variable assignment over V called the goal, and*
- *A is a finite set of (MPT) actions over V.*

A *partial (full) variable assignment* is a function f from $V' \subseteq V$ ($V' = V$) such that $f(v) \in D_v$ for all $v \in V'$. An *action* $\langle pre, \mathit{eff} \rangle$ consists of a partial variable assignment pre over V called its precondition, and a finite set of effects eff. Each effect contains a variable v called the affected variable, and $d \in D_v$ which represents the new value for v. We use $pre(a)$ to refer to the set of all variables that are in an action's precondition and $\mathit{eff}(a)$ to refer to the set of all variables belong to an effects of a.

Although different MPT encodings will lead to different agent decompositions of the problem, we do not discuss the details of their calculation here (see (Edelkamp and Helmert 1999)). We believe that, as long as the provided encoding attempts to minimise the number of variables over some sensible metric, it will be suitable for finding decompositions.

For our example problem, FD generates the following variables:

- Three variables that each represent the location of one of the robots (e.g. `a_loc`), each of which has five possible values,
- Three variables that represent whether or not each robot has reported (e.g. `a_rep`), each of which has two possible values,
- Five variables for whether or not each location is free (e.g. `x_free`), each of which has two possible values.

Along with the MPT representation of the problem, we also utilise the *causal graph* that FD creates. Causal graphs encode dependencies among variables based on the available actions.

Definition 2 (Causal graphs) *Let Π be a multi-valued planning task with variable set V. The* causal graph of Π, *$CG(\Pi)$ is the directed graph with vertex set V containing an arc (v, v') iff $v \neq v'$ and the following condition holds:*

- ***Transition condition*** *There is an action that can affect the value of v' which requires a value for v in its precondition.*

It is a fairly simple procedure to generate the causal graph from an MPT problem. Figure 2 shows the causal graph generated for our example problem.

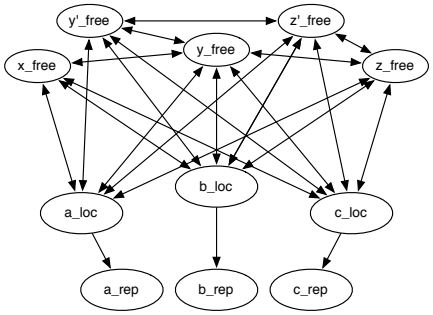

Figure 2: The causal graph for the problem in Figure 1.

5 Agent Decompositions

A good multiagent decomposition should split the domain so that as much as possible can be achieved independently and the amount of coordination between agents is minimised. We achieve this by finding sets of variables that can represent the internal states of agents. These internal variables cannot be changed by and are not required (as preconditions) by any other agents in the domain.

Even in easily decomposable domains there will be variables that are not internal to any agent. These are *public* variables and represent the environment that the agents are operating in. In our example domain, the variables that encode whether or not a particular location is free are public.

Definition 3 (Variable decomposition) *A* variable decomposition *of an MPT $\Pi = \langle V, I, G, A \rangle$ is a set $\Phi = \{\phi_1, \ldots, \phi_n\}$ along with a set $P = V \setminus \bigcup \Phi$ such that either:*

- *Φ is a* partition *of V (and therefore $P = \emptyset$), or*
- *$\Phi \cup \{P\}$ is a* partition *of V.*

Informally, a variable decomposition assigns a set of variables to each agent and a set of public variables. The variable sets cannot overlap. In practice there will be many public variables left over. For example, in the IPC Rovers domain (problem 20) there are 7 agents found (one for each rover object in the problem), 41 variables in total amongst the agent sets and 49 public variables. Even with this many public variables, we are able to still assign most actions to a single agent.

Definition 4 (Action Classification) *For an MPT $\Pi = \langle V, I, G, A \rangle$, variable decomposition Φ, agent i, and action $a \in A$:*
We call a an internal action *of i iff*

- $\exists v \in pre(a) : v \in \phi_i$ *and*
- $v \in pre(a) \rightarrow v \in \phi_i \cup P$.

We call a a joint action *of i iff*

- $\exists v \in pre(a) \ : \ v \in \phi_i$ *and*
- $\exists v' \in pre(a) \ : \ v \in \phi_j$ *with* $i \neq j$.

Finally, we call a a public action *iff*

- $v \in pre(a) \rightarrow v \in P$

We use Act_i to denote the set of all actions that are either internal or joint actions of i.

The set of *internal* actions for an agent is the set of all actions that require as preconditions at least one variable belonging to that agent's variable set and no variables belonging to any other agent. *Joint actions* are actions that require preconditions from multiple agents. These are generally the hardest to deal with in multiagent planning. *Public* actions are all those that only deal with environment variables. For the Rovers domain mentioned above we have 2004 internal actions, only 6 shared public actions, and no joint actions.

Definition 5 (Agent Variable Decomposition) *We call a variable decomposition Φ of MPT $\Pi = \langle V, I, G, A\rangle$ an* Agent Variable Decomposition *(AVD) or* agent decomposition *for short if $|\Phi| \geq 2$, there are no joint actions, and for all agents i and actions $a \in A$:*

$$\exists v \in \mathit{eff}(a) : v \in \phi_i \rightarrow a \in Act_i.$$

or equivalently:

$$\exists v \in \mathit{eff}(a) : v \in \phi_i \rightarrow \exists v' \in pre(a) : v' \in \phi_i$$

In other words, only actions belonging to a certain agents action set can affect the variables belonging to that agent. This is what we mean by an agent's variables being internal – there is no way for another agent to change them and they are never required as preconditions for another agent's action. Interaction occurs only through public variables.

The agent decomposition found for our example domain has each robot's location and whether or not it has reported as internal variables for that agent. Variables describing whether each grid-square is free or not are considered public variables common to all agents.

Finally, to help with classifying our decompositions we make the following further distinction:

Definition 6 (Influenced and Influencing Actions) *An action $a \in A$ is* influenced *if:*

$$\exists v \in pre(a) : v \in P$$

An action $a \in A$ is influencing *if:*

$$\exists v \in \mathit{eff}(a) : v \in P$$

The easiest actions to deal with are internal actions that are neither *influenced* or *influencing*: Influenced actions may require another agent to change the world before they can be used, influencing actions may change what other agents can achieve in the world. This means that internal actions can be either influenced, influencing, or both. Public actions are necessarily both influenced and influencing.

In our example domain, under the obvious decomposition, each `report` action is internal and neither influenced or influencing. Each `move` action is internal and both influenced and influencing because it relies on and changes the public `free` variables. An action that toggles whether or not a particular grid square is free would be a public action.

6 Agent Decomposition Algorithm

The *agent decomposition-based planner* (ADP) we propose has two parts: In a first step, it calculates an agent decomposition for the given MPT. This does not require plan search and is based on a static analysis of the causal graph created by FD. In a second step, it uses the decomposition of the problem to search for a plan. As not all domains are multiagent the first step may return that no decomposition can be found. In this case, our search algorithm defaults to a standard single-agent algorithm.

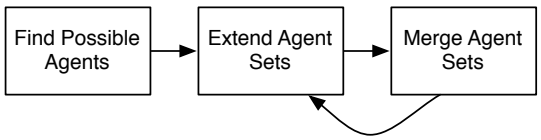

Figure 3: Overview of the agent decomposition algorithm

Algorithm 1: Agent Decomposition Algorithm

Input : MPT $\Pi = \langle V, I, G, A\rangle, \Phi = \emptyset$

1) Find Possible Agents
foreach *Variable* $v \in V$ **do**
  if $\forall (v', v) \in CG \ : \ (v, v') \in CG$ *and* $\exists (v, v'') \in CG : (v'', v) \notin CG$ **then**
    $\Phi \leftarrow \Phi \cup \{v\}$
if $|\Phi| < 2$ **then**
  return $\emptyset$

repeat
  2) Extend Agent Sets
  foreach *Agent set* $\phi_i = \{v_1, \ldots, v_n\} \in \Phi$ **do**
    foreach $v_s \in V : v_s$ *is an internal successor of an element of* ϕ_i **do**
      $\phi_i \leftarrow \phi_i \cup v_s$

  3) Merge Agent Sets
  foreach *Pair of agent sets* $\phi_i, \phi_j \in \Phi$ **do**
    if $\phi_i \cap \phi_j \neq \emptyset \vee \exists$ *a joint action of both* i *and* j **then**
      Merge ϕ_i and ϕ_j into one agent
  if $|\Phi| < 2$ **then**
    return $\emptyset$
until Φ *is unchanged*

return Φ

We discuss the decomposition algorithm first, which is split into three parts as shown schematically in Figure 3. The first part finds variables that are candidates to become members of an agent set. The second part of the algorithm extends the found nodes to their neighbours maximising the number of internal variables for each agent. The third part of the algorithm combines agents to remove joint actions and merge agents with overlapping variable sets. If the second and third parts change the decomposition then they are repeated. Any leftover variables belong to the public variables set.

1) Find Possible Agents This part of the algorithm is shown at the top of Algorithm 1. Looking back at the causal

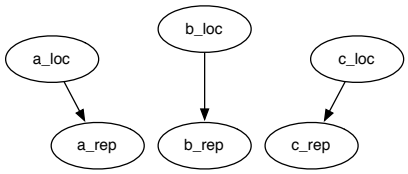

Figure 4: The causal graph for our example problem with cycles removed

graph definition we can see that root nodes in the causal graph (if they exist) have the property that they do not depend on the value of other variables. Taking each root node of a causal graph as a separate agent set always creates an agent decomposition when there are two or more root nodes. However, as can be seen from Figure 2, even in very simple domains, it may be that no root nodes exist.

If there are no root nodes in the causal graph then it must contain cycles. Cycles of dependencies in causal graphs are a common complication for many planning approaches. Fast Downward for example cannot deal directly with cycles, and employs a heuristic method to remove certain edges to make the causal graph acyclic. In our case, we only need to remove cycles of order 2 from the graph to ensure that all possible agents become root nodes. The causal graph of our example problem with 2-way cycles removed is shown in Figure 4.

The algorithm checks each variable to see if it would be a root node once all 2-way cycles are removed from the graph. If that variable has at least one successor left (ie. it is not completely disconnected from the rest of the graph) then it is added to Φ as a separate agent. If no root nodes are found by this method then there is no possible agent decomposition.

2) Extend Agent Sets This part of the algorithm extends the agent sets we have so far to make them as large as possible.

Definition 7 (Internal Successors) *We call a variable v an internal successor of variable set V if v is a successor of a member of V in the causal graph and all predecessors of v are in V.*

The operation of expanding the agent sets like this preserves the agent variable decomposition property. However, at this stage of the algorithm it is still possible to have a decomposition that is not an agent decomposition.

3) Merge Agent Sets The previous part of the algorithm can create agents that share variables. The first merging step combines those agents into one. It is also possible that there are joint actions under our decomposition. The second merging operation combines agents that share joint actions.

Theorem 6.1 *The algorithm presented in this section for finding an agent decomposition given an MPT is both sound and complete.*

Sketch of Proof The proof of this uses the following steps and relies heavily on the link between the transition condition definition for causal graphs and the AVD property. In the case where all variables found in Part 1 are root nodes in the original graph then Part 1 produces an agent decomposition and parts 2 and 3 preserve this. In the other case for any pair of sets that violate the agent decomposition property it can be shown that there must be a path between them and that they will be merged in Part 3. Finally, when an agent decomposition exists it can be shown that there must be at least two variables found in Part 1 that will not be merged in Part 3 of the algorithm.

Example In our example domain, ignoring 2-way cycles results in the graph shown in Figure 4. The variables corresponding to robot locations are root nodes of this graph and Part 1 creates the set $\{\{\texttt{a\_loc}\},\{\texttt{b\_loc}\},\{\texttt{c\_loc}\}\}$. Part 2 extends the agent sets to $\{\{\texttt{a\_loc, a\_rep}\},\ldots,\{\texttt{c\_loc, c\_rep}\}\}$. The third part does not alter the decomposition. The remaining variables are public variables for the domain.

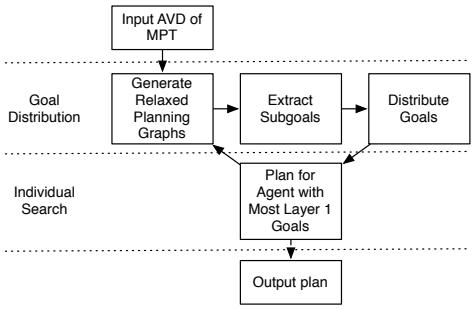

Figure 5: Overview of the search component of ADP.

7 Plan Synthesis Algorithm

We now turn to the plan synthesis component of ADP. For plan synthesis, we use forward state-space search and utilise the common "no delete lists" heuristic (Hoffmann and Nebel 2001). We only search for a single agent in all parts of the algorithm. To achieve this, we use a goal distribution method to identify which agent should be used next, and what goals it needs to achieve. These goals might include propositions that are needed to help other agents to find the overall solution, but which are not members of the final goal set.

The full search algorithm proceeds in iterations of goal distribution and plan search as outlined in Figure 5. In every iteration, goal distribution determines a set of useful propositions that can be solved from the current state by a single agent. FF heuristic search is then used (on the relevant agent's subproblem only) to find a plan that achieves those propositions. This results in a new state. The process is repeated from this new state until a plan has been found or a dead end is reached.

Any plan search conducted as part of this algorithm only considers *individual planning problems*. An agent ϕ_i's individual planning problem is the planning problem made up of only the variables in $\phi_i \cup P$. In other words, a plan search for agent i only considers actions in Act_i and public actions.

Goal Distribution

The aim of goal distribution is to find the next set of propositions to achieve. This is split into three parts: First, relaxed planning graphs are generated for each agent until all goal propositions have been reached. In the second part, relaxed plans are extracted in order to find out which propositions need to be achieved to traverse between agents' individual problems. These become subgoals for our problem. The third step distributes the goals and subgoals among the agents that can achieve them.

Generating Relaxed Planning Graphs We start by generating the internal relaxed planning graph for every agent ϕ_i from the current state. This relaxed planning graph (using the no-delete-lists relaxation) is generated by repeated application of elements of Act_i and public actions.

Algorithm 2: Search Algorithm

```
repeat
    Generate Planning Graphs
    S ← current state
    G ← original goal set
    repeat
        foreach Agent φ_i do
            Generate Internal Relaxed Planning Graph for φ_i
            from S
        S ← S ∪ the final state of each relaxed planning
        graph.
    until All goals reached OR no new propositions added
    if Not all goals reachable then
        return dead end

    Extract Subgoals
    foreach g ∈ G do
        Extract relaxed plan for g
        if layer(g) > 1 and p is required by previous layer
        then
            G ← G ∪ p
            Extract relaxed plan for p

    Distribute Goals
    foreach g ∈ G : layer(g) == 1 do
        Add g to G_i for agent with lowest estimated cost

    Agent Search
    φ_i ← φ_i ∈ Φ such that |G_i| is maximized
    FF search φ_i's subproblem with G_i as goal set
    Current state ← result of φ_i's plan applied to S
until All goals reached or dead end found
```

Naturally, some of the propositions reachable in the overall planning problem may not appear in any of the agents' internal relaxed planning graphs, as their achievement may requite interaction among the agents. For example, in the problem shown in Figure 1, robot `b` cannot perform any actions from the initial state even though it can clearly reach its goal once another robot moves out of the way.

To share states between agents efficiently, we simply combine all propositions reached in the individual planning graphs. This creates a new state which can be used as the input for a subsequent layer of individual planning graphs. From this new state it may be possible to reach new areas of the state space. By repeating this process as many times as required, it is possible to reach any proposition that is reachable in the full planning problem.

Definition 8 (Planning Graph Layer) *Given a state S, planning graph layer 1 contains each agent's individual planning graph starting from state S. Each subsequent layer*[2] *contains each agent's individual planning graph starting from the combined states of all graphs in the previous layer. We use $layer(p)$ to represent the layer in which proposition p first appears.*

In our example problem, `free_y` is false initially, so robot `b` cannot move into this space. In the internal planning graph for robot `a` from layer 1, a `move` action sets `free_y` to true. Because we are using the no delete lists heuristic, the value of `free_y` is now both true and false. This means that in the second layer of planning graphs robot `b` can move into this space.

Planning graphs are built until all unreached goal propositions are found or no new actions can be performed. If the full goal set has not been reached, then we have reached a dead end in the search space. For goal propositions that only appear beyond layer 1, plan extraction can be used to determine the input that is required from other agents.

Extracting Subgoals Plan extraction is used in FF to generate relaxed plans for calculating heuristic values. It proceeds by picking a goal proposition, and then working backwards through a relaxed planning graph. Preconditions are found for an action that adds the goal proposition. These preconditions are then added to the list of propositions to extract from. Eventually, plan extraction leads back to propositions that only appear in the starting state.

In our case, if plan extraction is performed on a proposition added in any layer beyond the first one, we will end up at the root state of the planning graph corresponding to that layer. This may include propositions from the initial state but also propositions added by other agents. Whenever a proposition added by another agent is reached, we add it to the goal set and extract a relaxed plan to reach this proposition.

We call propositions that are added to the goal set in this way *subgoals*. As we only ever search one agent's subproblem at a time and are therefore limited to propositions reachable in the first layer, we only care about subgoals that are reachable in layer 1.

Distributing Goals In this phase of the algorithm, as we want to identify the most suitable agent for every goal reachable in layer 1 (including newly generated subgoals). We assign each goal to the agent with the lowest estimated heuristic value based on their extracted relaxed plans. The choice of which agent to plan for next is then made based on which agent appears has greatest number of goals in layer 1. Any goals that have already been achieved are included in the goal list to make sure they are not undone during agent search.

Agent Search We use standard FF planning to solve that agent's subproblem from the current state to a goal state where all of its layer 1 goal propositions hold. We additionally require that a plan cannot end up in a state that has already appeared. This prevents cycles between agents from occurring. The entire process is repeated from the state reached by agent search.

It may be possible to create a subset of goals that are not solvable from the current position. It is an open research question what the best method for solving this problem is. In our current implementation we join the agents together, thus effectively reverting to the original planning problem. In this case, our search method behaves roughly equivalent to normal FF.

[2]Note that the term layer here should not be confused with layers in relaxed planning graphs. When used in this document it refers to a collection of individual relaxed planning graphs that have all started from the same state.

Domain Name	Agents Min	Agents Max	Agent Var %	Percentage of Each Action Type I	$> I$	$I <$	$> I <$	P	Decomposition Variables
Airport	2	15	0.5	50	0	5	45	0	(Plane locations)
Depot	3	7	28	48	0	9	41	2	(Truck locations) + (all Crate locations)
Driverlog	2	6	18	46	0	17	33	4	(Truck locations)
Elevators	4	5	17	50	1	0	49	0	(Lift locations)
Logistics	3	7	34	50	4	0	46	0	(Truck locations)+(Plane locations)
Rovers	2	14	43	52	33	0	15	0	(Rover locations, Calibrated and Images)
Satellite	2	12	72	51	49	0	0	0	(Satellite locations, Calibrated, Images and Instruments)
SatelliteHC	5	15	38	50	50	0	0	0	(Satellite locations, Calibrated, images and instruments)
Tpp	2	8	3	50	0	0	50	0	(Truck locations)
Transport	4	4	15	50	1	0	49	0	(Truck locations)
Zenotravel	2	5	39	50	37	0	12.8	0	(Plane locations, Fuel-Level)
Floortile	2	3	7	13	0	0	50	37	(Robot has x)
Freecell	2	6	38	34	0	10	40	16	(Home suits) + (large collection of other variables)
Mprime	2	16	15	33	0	2	48	18	(Craves(X, Y))
Pathways	6	66	21	36	0	18	32	14	(Chosen(X))
Woodwork	6	85	12	23	0	1	49	27	(Colours) + (all Available and saw)

Table 1: Decomposition results for IPC domains that returned agent decompositions. Agent Var % is the percentage of variables in the domain that are in an agent's variable set. I represents internal actions. We use the shorthand $>$ to mean influenced and $<$ to mean influencing. The $>$I$<$ column therefore lists the percentage of both influenced and influencing internal actions. P are public actions.

Solving Our Example Problem

In our example, robot `a` can reach the goal `report_a` from the initial state. The goals `report_b` and `report_c` are only reachable in layer 2. Plan extraction adds the subgoal `free_y` to agent a's goal set. The first plan to be found is therefore one for agent a that reaches `report_a` and `free_y`. The plan that achieves this moves `a` one square to the left and then `a` performs its report action.

From this new state, robots `b` and `c` still cannot reach square `x` to report in layer 1 of their planning graphs. Plan extraction again reveals that robot `a` needs to move out of the way. As this is the only achievable layer 1 goal then this is solved next. Eventually we reach a state where robot `b` can reach the goal `report_b` in layer 1. From here a similar process is repeated for robot `c` to be able to achieve `report_c`.

8 Results

To evaluate the performance of our method, ADP was implemented as an extension of the Fast Downward planning system (Helmert 2006). The translation and preprocessing steps were left unmodified and ADP was added as an alternative search option. We ran ADP over every domain included in FD's benchmarks, using the 2008 Satisficing Track version whenever multiple instances were available.

Decomposition Results

The results of applying our decomposition algorithm are shown in Table 1 for those domains for which a decomposition was found at least for some instances.

The top half of the table lists domains for which a decomposition was found in all but the simplest problem instances. Some of the the simplest instances for each domain contain only one of the objects that would be normally identified as an agents. For example, the first few Rovers problems contain only one rover. In cases like these, the algorithm returns no decomposition. This is the behaviour we would expect as these are effectively single-agent problems.

The bottom half of the table contains domains where a decomposition could only be found for a few instances of the problem. As can be seen in the table, these decompositions contain large numbers of public actions and both influenced and influencing internal actions. As expected, these are domains in which the plan synthesis part of our method does not perform well compared to planning without decomposition.

The first two columns are included to give an idea of the numbers of agents we deal with. Smaller problem instances often contain less agents than larger ones. As the size of problems increase, more agents are added, yet the action ratios remain relatively stable. To exemplify this, the Satellite is split into two separate rows (satellite and satelliteHC). As can be observed, the ratios of different action types are roughly the same across both subsets of instances.

The percentage of purely internal (neither influenced nor influencing) actions provides a good estimate of how well a multiagent approach will perform. As we can see, these actions account for around 50% of all actions for most easily decomposable domains. Influenced only internal actions ($>$I column) are the next easiest to plan with as no other agents are affected when these are performed. This means that they can be used freely when solving an individual planning problem, as they they cannot undo goals that others have already been working towards.

Generally speaking, the decompositions found are the ones a human designer would expect: In the Airport domain, agents are planes, in Driverlog agents are trucks, in Elevators lifts become agents, etc. The Logistics domain is separated into one agent for each truck and one for each airplane, and agents in Satellites are the satellites combined with their respective instruments and other apparatus.

	Max	Coverage			Search Time (s)			Cost			States Expanded $\times 10^3$			Ratio	
	Layer	ADP	FF	LM	ADP	FF	LM	ADP	FF	LM	ADP	FF	LM	ADP	PB
Airport	3	**19**	17	15	**0.6**	66	1.8	1080	1078	**1066**	**1.1**	92	2.4	**110**	0.96
Depot	3	19	18	**20**	214	237	**138**	1109	1128	**890**	578	724	**127**	**1.1**	1.03
Driverlog	1	20	20	20	83	60	**24**	1497	1520	**1307**	635	447	**127**	0.7	**1.2**
Elevators	2	12	10	**14**	0.54	316	**0.37**	2072	2504	**1879**	13.4	4865	**3**	**585**	–
Logistics	3	28	28	28	**0.0**	0.1	0.1	**1184**	1189	1190	**1.9**	5.0	2.2	1	**2.15**
Rovers	1	38	38	38	**2.0**	21	61	4300	4484	**4233**	**9.2**	51.9	76.9	**10.5**	2.61
Satellite	1	18	18	18	**0.2**	0.3	1.6	892	797	**751**	**3.0**	3.7	12.9	1.5	**13.7**
SatelliteHC	1	**13**	11	11	**11.5**	102	38	1938	2013	**1693**	**12.5**	28.3	19.3	8.9	**13.7**
Tpp	1	25	25	25	**13.4**	28.0	71.4	4038	3641	**3482**	**27.3**	28.8	43.5	**2.1**	1.03
Transport	1	**18**	14	15	**0.09**	3.57	0.13	3047	3812	**2188**	**308**	11976	343	**40**	1.5
Zenotravel	1	18	18	18	**0.6**	2.7	4.4	741	**673**	683	**2.3**	7.3	5.1	**4.5**	2.7
Average	1	**20.7**	19.7	20.2	**29.6**	76.1	31.0	1991	2076	**1760**	145	1657	**69**	–	–

Table 2: Results comparing ADP to FF and LAMA on the IPC domains for which it found a good decomposition. Max Layer shows the maximum number of layers needed for ADP's generate planning graphs step. Coverage shows the number of successfully solved problems. The middle columns show the sum over all the problems that were successfully solved by all three planners. The right hand columns are comparison between ADP and (Nissim, Apsel, and Brafman 2012).

Search Results

Alternative multiagent planning approaches follow distinct objectives, making it problematic to make direct comparisons. As FF (Hoffmann and Nebel 2001) and LAMA (Richter and Westphal 2010) are competitive satisfying planners based on a similar implementation to ADP we choose to evaluate ADP against these two algorithms. We use the implementation of FD as this shares large amounts of code with ADP. All experiments reported below were run on the same 2.66GHz machine and every planner was given a maximum of five minutes to solve each problem.

Table 2 shows the results of our empirical analysis. We present search time instead of total time because the preprocessing process is the same for all three algorithms. In fact, we only need to run the preprocessing once per problem and can use its output as the input for each of the three planners. The agent decomposition part of ADP is included as part of the search time of the ADP algorithm but is negligible compared to the time needed to perform plan search.

In this table, the "coverage" column shows the number of problems solved by each algorithm within the time limit. ADP solves more problems than the other algorithms on Airport, Satellite and Transport, while only solving fewer problems than the other algorithms in Depot and Elevator. For most of the domains, however, all multiagent problems are solved by all planners and it is hard to predict what would happen with larger instances.

The rest of the table gives averages over only the problems solved by all three planners. The algorithm that expands the smallest number of search states is always the fastest for that problem. This suggests that the goal decomposition stage of ADP is not very expensive as it is called many times during search and does not increase the number of expanded states directly. LAMA has the lowest average number of expanded states despite only being better in three of the eleven domains. If we were to choose to only use a decomposition when no public actions at all exist, then ADP would be clearly ahead of its competitors.

ADP tends to find longer plans than LAMA but shorter plans than FF. This suggests that it would be worthwhile for the algorithm to spend more time on analysing the relaxed planning graphs created during the goal distribution phase. A more elaborate goal distribution method might lead to better subgoals being generated, and improve planning times by reducing the number of states that need to be expanded.

As mentioned in Section 2, an approach that is closely related to ours is the partition-based pruning (PBP) approach presented by (Nissim, Apsel, and Brafman 2012). Unfortunately, the two algorithms do not lend themselves to a direct comparison of planning times or costs due to the different nature of the algorithms: Partition-based pruning is part of an optimal planning algorithm and therefore returns plans of lower cost, albeit at the expense of much longer planning times than ADP. To overcome this incomparability, we look at the the speedup that both methods provide compared to their respective single-agent versions (A* and FF). This is shown in the rightmost columns of Table 2.

Nissim et al's decomposition method finds decompositions for many more domains than ADP. This is because PBP does not require decompositions to be agent variable decompositions. However, the only domains for which decompositions help speed up their search but don't help for ADP are Driverlog and Pathways. On the other hand, our approach is more effective in Airports, Elevators and Transport.

9 Conclusion

In this paper, we have presented a new decomposition method that breaks down classical STRIPS-style planning domains into a set of individual sub-problems for different agents, and a novel planning algorithm that exploits the additional structure afforded by such decompositions.

Our empirical evaluation shows that around one third of the IPC domains are suitable for our approach. For these domains, our ADP algorithm is shown to outperform state-of-the-art planners in the majority of cases. In domains that require some interaction between the agents it outperforms its single agent counterpart by many orders of magnitude.

References

Boutilier, C., and Brafman, R. 2001. Partial-order planning with concurrent interacting actions. *Journal of Artificial Intelligence Research* 14:105–136.

Brafman, R., and Domshlak, C. 2008. From One to Many: Planning for Loosely Coupled Multi-Agent Systems. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 18, 28–35.

Brafman, R.; Domshlak, C.; Engel, Y.; and Tennenholtz, M. 2009. Planning Games. In *Proceedings of the International Joint Conference on Artificial Intelligence*, volume 21, 73–78.

Brenner, M., and Nebel, B. 2009. Continual planning and acting in dynamic multiagent environments. *Journal of Autonomous Agents and Multi-Agent Systems* 19:297–331.

Cox, J., and Durfee, E. 2005. An efficient algorithm for multiagent plan coordination. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems*, volume 4, 828–835.

Dimopoulos, Y.; Hashmi, M. A.; and Moraitis, P. 2012. μ-satplan: Multi-agent planning as satisfiability. *Knowledge-Based Systems* 29(0):54 – 62.

Edelkamp, S., and Helmert, M. 1999. Exhibiting knowledge in planning problems to minimize state encoding length. In *ECP*, 135–147.

Ephrati, E., and Rosenschein, J. 1993. Multi-agent planning as the process of merging distributed sub-plans. In Katia P. Sycara, M. F., ed., *In Proceedings of the Twelfth International Workshop on Distributed Artificial Intelligence (DAI 1993)*, 115—129.

Helmert, M. 2006. The fast downward planning system. *Journal of Artificial Intelligence Research* 26:191–246.

Hoffmann, J., and Nebel, B. 2001. The ff planning system: Fast plan generation through heuristic search. *J. Artif. Intell. Res. (JAIR)* 14:253–302.

Lansky, A. 1991. Localized search for multiagent planning. In John Mylopoulos, R. R., ed., *Proceedings of the Twelfth International Joint Conference on Artificial intelligence (IJCAI 1991)*, 252–258.

Nissim, R.; Apsel, U.; and Brafman, R. I. 2012. Tunneling and decomposition-based state reduction for optimal planning. In Raedt, L. D.; Bessière, C.; Dubois, D.; Doherty, P.; Frasconi, P.; Heintz, F.; and Lucas, P. J. F., eds., *ECAI*, volume 242 of *Frontiers in Artificial Intelligence and Applications*, 624–629. IOS Press.

Nissim, R.; Brafman, R.; and Domshlak, C. 2010. A general, fully distributed multi-agent planning algorithm. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems*, volume 9, 1323–1330.

Richter, S., and Westphal, M. 2010. The lama planner: Guiding cost-based anytime planning with landmarks. *J. Artif. Intell. Res. (JAIR)* 39:127–177.

Tonino, H.; Bos, A.; de Weerdt, M.; and Witteveen, C. 2002. Plan coordination by revision in collective agent based systems. *Artificial Intelligence* 142(2):121–145.

Yang, Q.; Nau, D.; and Hendler, J. 1992. Merging separately generated plans with restricted interactions. *Computational Intelligence* 8(4):648–676.